Bridging the Gap Between COTS Product Reuse and Formal Methods: A Case Study

Edward A. Addy
NASA/WVU Software Research Laboratory
100 University Drive
Fairmont, WV 26554
eaddy@wvu.edu

Murali Sitaraman
Computer Science and Electrical Engineering
West Virginia University
Morgantown, WV 2650-6109
murali@csee.wvu.edu

Abstract
Reusable commercial off-the-shelf (COTS) products are routinely employed in development of software systems. However, no systematic techniques are available for specification or verification of critical aspects of such systems. This paper explains that the dependencies between a critical subsystem and a COTS product can be isolated through formally-stated mathematical and programmatic interface contracts. The contracts allow specification and reasoning of critical subsystems, without a need to describe entire COTS product functionality formally. They also provide the flexibility of using alternative COTS products that include the desired behavior.

The paper illustrates elements of the proposed approach using a subsystem of the NASA/FAA Surface Movement Advisor (SMA). This case study is based on a COTS database.

1. Introduction

Developers of large software systems are turning increasingly to the use of commercial software developed by third parties [Voas 98a, Weyuker 98]. Use of COTS[1] software as a large-grain component of a new system allows the developer to reduce time to market and to reduce system development costs. COTS software is almost always delivered in executable form and rarely is there direct access to the requirements or design documentation. When COTS software is employed in life-critical, mission-critical, or in any system with significant financial liabilities, it is essential to ensure that the integrated system is reliable.

Typically, it is neither possible nor feasible to develop complete and formal descriptions of behaviors of COTS products. But unless the behaviors are described formally, it is not possible to use rigorous techniques for specification or reasoning of COTS-based safety-critical systems. The contributions of this paper are in addressing this dilemma. It describes a solution that is based on complete specification of partial functionality of COTS products, and illustrates the solution approach using a realistic case study.

This paper explains that a fundamental issue to be tackled in using a COTS product is the specification of the safety-critical subsystem. To the extent required in mathematical modeling and specification of the critical subsystem, features of the COTS product need to be modeled mathematically and described in *mathematical interface contract(s)*. To perform formal reasoning or rigorous validation of the implementation of the COTS-based subsystem, *programmatic interface contract(s)* or specification of the commercial product or legacy system is necessary. The interface contracts isolate and precisely describe those aspects of the COTS product that affect the application system. In addition to enabling unambiguous, understanding and formal reasoning, precise descriptions of the interfaces also guide the testing that must be performed on the COTS product and integration testing.

[1] The terms COTS and commercial software are used to reference software developed by a third party. Much of the discussion in this paper also applies to any software previously developed outside the application development environment, including legacy and public domain software, and other non-developmental items.

The case study, on which this paper is based, concerns a subsystem of the Surface Movement Advisor (SMA) system, a joint Federal Aviation Administration (FAA) and National Aeronautics and Space Administration (NASA) effort [SMA 95]. The objective of the SMA is to assist air-traffic controllers in the area of ground movement control. SMA uses a COTS database product, among others. Our experience in specifying and implementing a subsystem of SMA demonstrates that mathematical modeling of realistic software subsystems is possible, even in the presence of significant COTS software usage.

The rest of the paper is organized into the following sections. Section 2 of the paper contains a brief overview of the SMA. Section 3 contains a formal specification of an SMA subsystem, based on a partial mathematical modeling of the COTS product behavior. Section 4 outlines a component-based implementation of the subsystem. It contains partial programmatic interface contracts for isolating COTS product interaction. Section 5 summarizes related work, status of the case study, and our conclusions.

2. Surface Movement Advisor

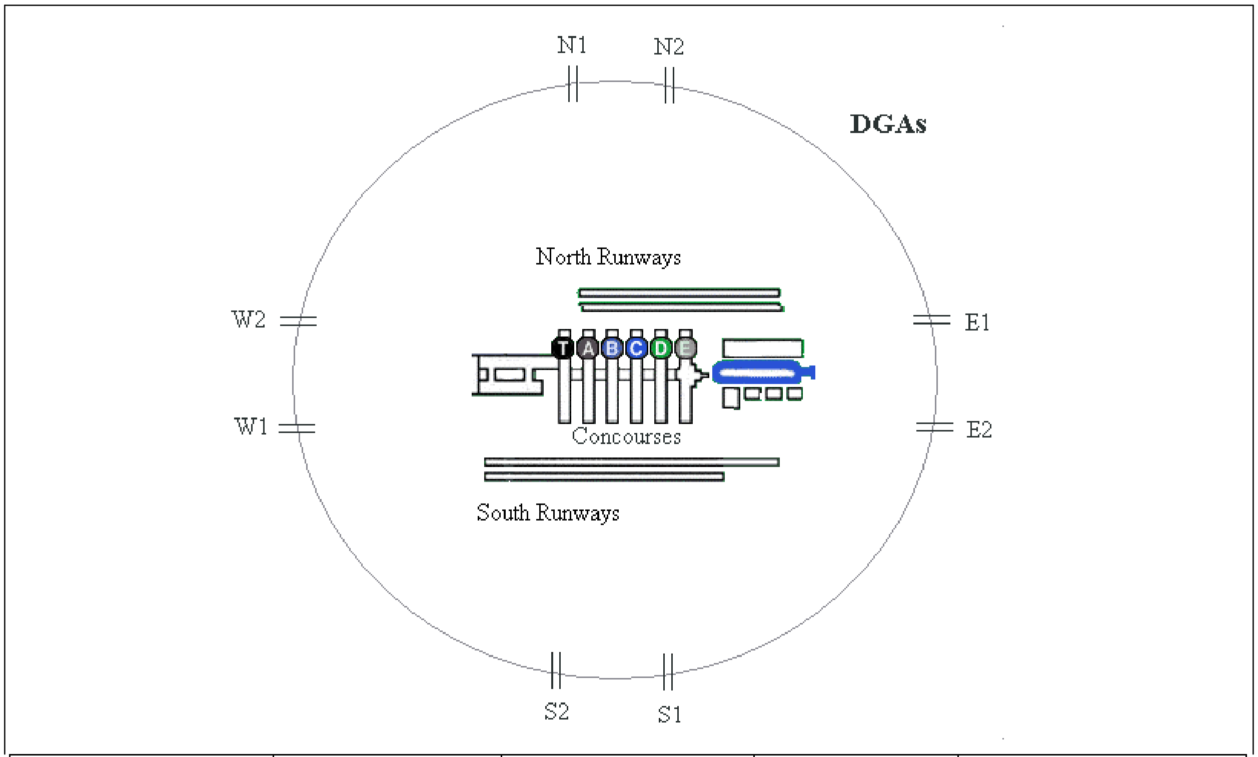
SMA is a proof-of-concept prototype demonstration to test the implementation of advanced information systems technologies to improve coordination and planning of ground airport-traffic operations. SMA is primarily a data fusion and dissemination system, integrating airline schedules, gate information, flight plans, radar feeds and runway configuration (including departure split and landing direction). This integrated information is shared among airport ramp operators, managers, airline operators, and FAA controllers and supervisors. As a prototype system, SMA currently is targeted for the Atlanta Hartsfield International Airport. Following a successful demonstration, the system will be modified for installation at other major airports.

The subsystem of SMA considered in this paper deals with prediction of the times of key events for flights. This subsystem is responsible for determining the most likely time for flight events such as time of pushback from the gate, takeoff time, landing time, and gate arrival time.

The case study is a shadow development effort, based on the SMA Systems Requirements Document, Build 1, of SMA [SMA 95]. The focus of the study is the "departure part" of the prediction subsystem that is concerned with predicting flight takeoff times. All departing flights leave the airspace surrounding an airport through one of several designated points, called Departure Gate Areas (DGAs). The DGAs can be conceived as horizontal tunnels in the sky through which the flight must pass. The DGA that a flight will use is determined by the destination of the flight.

Each airport has a set of standard configurations or "splits", that determines the runway that a flight will use based on the DGA of the flight. The airport has a number of pre-defined splits, but splits may also be created on an ad-hoc basis. The DGAs and some of the pre-defined splits at Atlanta Hartsfield International Airport are depicted in Figure 1. A key goal of the air-traffic controller is to minimize the time that the planes spend after pushback waiting to take off. This is done by attempting to keep the runways balanced in terms of utilization.

To make predictions on flight departures, SMA needs access to a large amount of flight and airline information. This information is managed using an Oracle database, a COTS product. The database contains (among other information) the scheduled, predicted, and actual pushback time for each flight, the call number for the plane, the aircraft type, the boarding gate, and the DGA the flight will use. The users can display the cumulative wait times for each runway, one split at a time, or can display a graphical comparison of all the splits. (The display can show the cumulative wait times for any window between 15 and 60 minutes, beginning at the current time.)



SPLIT	NW/SE	NWE1/SE2	NE/SW	N/SWE
DGAs using the north runway	N1, N2, W1, W2	N1, N2, W1, W2, E1	N1, N2, E1, E2	N1, N2
DGAs using the south runway	S1, S2, E1, E2	S1, S2, E2	S1, S2, W1, W2	S1, S2, W1, W2, E1, E2

Figure 1. Departure Gate Areas and Several Predefined Splits for Atlanta Hartsfield International Airport (conceptual drawing, not to scale)

3. Formal Specification of the Flight Takeoff Prediction Subsystem

The Prediction Subsystem is responsible for calculating the cumulative wait times and the predicted takeoff time for each flight, given the runway, the split to be used, and the time window. The focus of the problem is predicting the time of takeoff for a flight, taking into account the time the flight pushes back from the gate and considering other flights that might impact the time of takeoff.

An object-based specification for the takeoff time prediction subsystem, named Simulate_Runway_Machine_Template, is shown in RESOLVE notation [Sitaraman 94] in Figure 2. Other formal notations, such as Larch, VDM, or Z, summarized in [Wing 90] could have been used equally well.

Simulate_Runway_Machine_Template provides a prediction simulation type, and operations to manipulate objects of the type. It is a result of "recasting" the prediction algorithm as an object. The idea of recasting graph and simulation algorithms as object-based machines, and the performance and software engineering benefits of the idea are discussed elsewhere [Weide 94a].

The RESOLVE **concept** or specification in Figure 2 contains a **context** section showing a list of "imports" and an **interface** section of "exports". In the figure, the **global context** imports Standard_Partial_SMA_Departure_ Database_ Facility, because the current specification relies on the mathematical modeling of database attributes described in that module. The **local context** contains mathematical definitions or **math operations** (explained later) that make it possible to write assertions in the interface section concisely.

```
concept Simulate_Runway_Machine_Template

  context

    global context

      facility Standard_ Partial_SMA_Departure_
        Database_Facility

    local context

-- mathematical definitions of
-- In_Line, Cumulative_Wait_Time_Def,
-- Safety_Delays_Met, Actual_Takeoff_Times_Used,
-- and Taxi_Times_Met

      math operation Well_Formed_Runway_Queue (
          q: string of FLIGHT_TIME_PAIR,
          sid: SPLIT_ID,
          rid: RUNWAY_NAME,
          db: SMA_DEPARTURE_DB_MODEL) : boolean
      explicit definition
        for all ft: FLIGHT_TIME_PAIR
          where (db.runway_used(ft.f) = rid or
            (db.runway_used(ft.f) = empty_string and
             db.assigned_runway
                       (sid,db.departure_gate_area(ft.f))   = rid)
          (In_Line(q, ft.f, db))
        and Safety_Delays_Met(q, rid, db)
        and Actual_Takeoff_Times_Used(q, db)
        and Taxi_Times_Met(q, rid, db)

      math operation Proper_Flights_In_Queue (
          sid: SPLIT_ID,
          rid: RUNWAY_NAME,
          tbegin: DISCRETE_TIME,
          tend: DISCRETE_TIME,
          q: string of FLIGHT_TIME_PAIR,
          db: SMA_DEPARTURE_DB_MODEL) : boolean
      explicit definition
        for all ft: FLIGHT_TIME_PAIR
          where IS_ENTRY_OF(q, ft) (
          tbegin <= ft.t <= tend)
        and
        there exists c: CONFIGURATION,
          q1: string of FLIGHT_TIME_PAIR
          where (db.c.id = sid and
            Well_Formed_Runway_Queue(q1, sid, rid, db)) (
           (Is_Substring (q, q1) )

 interface

   type Simulate_Runway_Machine_State is (
         sid: string of character,
         rid: string of character,
         tbegin:  DISCRETE_TIME,
         tend:  DISCRETE_TIME,
         q: string of FLIGHT_TIME_PAIR,
         ready_to_extract: boolean)
     exemplar m
     constraints
       Is_Allowed_Split_Name (m.sid) and
       Is_Allowed_Runway_Name (m.rid) and
       m.tend >= m.tbegin
     initialization ensures m.ready_to_extract = false

-- operations to set and obtain the values of the
-- split ID, the runway ID, and the beginning and
-- ending times, and an operation to check
-- if the machine is in extraction phase.

   operation Simulate_Runway (
       alters   m: Simulate_Runway_Machine_State,
       preserves db: SMA_Database_Machine)
     requires m.ready_to_extract = false
     ensures Proper_Flights_In_Queue
                   (m.sid, m.rid, m.tbegin, m.tend, m.q,db)
       and m.sid = #m.sid and m.rid=#m.rid and
       m.tbegin=#m.tbegin and m.tend=#m.tend and
       m.ready_to_extract=true

   operation Extract_Next (
       alters   m: Simulate_Runway_Machine_State,
       produces  flight_number: Char_String,
       produces  takeoff_time: Integer)
     requires |m.q| /= 0 and m.ready_to_extract = true
     ensures #m.q = <flight_number, takeoff_time> * m.q
       and m.sid=#m.sid and m.rid=#m.rid and
       m.tbegin=#m.tbegin and m.tend=#m.tend and
       m.ready_to_extract=#m.ready_to_extract

   operation Cumulative_Wait_Time (
       preserves m: Simulate_Runway_Machine_State,
       preserves db: SMA_Database_Machine,
       produces wait_time: Integer)
     ensures Cumulative_Wait_Time =
       Cumulative_Wait_Time_Def (m.q, m.rid, db)

   operation Queue_Length_Of (
       preserves m: Simulate_Runway_Machine_State,
       produces  length: Integer)
     requires m.ready_to_extract = true
     ensures length = |m.q|

end Simulate_Runway_Machine_Template
```

Figure 2: Flight Departure Prediction Concept

In the interface of this concept, objects of the type Simulate_Runway_Machine_State are *modeled* mathematically as a 6-tuple: sid and rid, respectively, denote the split identification and runway identification for which simulation is to be done. tbegin and tend denote the window for simulation. The central part of the model is the q that contains results of the simulation. It is a string (or sequence) of ordered pairs denoting which flight is predicted to depart at what time. The purpose of the boolean ready_to_extract will become clear in the following discussion. When it is true, conceptually, results from simulation are available for extraction. The specification also states, using an **exemplar** object, the **constraints** and **initialization** guarantees on every object of the type. Notice that initially ready_to_extract boolean is false.

The interface provides following operations on prediction simulation objects:

- operations to set/get simulation parameters (split, runway, start time, and end time);
- an operation to instruct the machine to simulate;
- an operation to extract the flight number and time at which the next flight is predicted to take off ;
- an operation to get cumulative wait time for the current simulation; and
- status-checking operations to see if the information on the next flight can be extracted and if there are any more flights for take off in the simulated window.

In a typical use of the object, after simulation parameters are set, the Simulate_Runway operation will be called. This operation uses and **preserves** the database, but **alters** the machine state. It **requires** that the machine be in the insertion phase, i.e., m.ready_to_extract boolean must be false. For the operation to work as specified in the ensures clause, the requires clause must hold when it is called.

The conceptual effect of calling the Simulate_Runway operation is that it **ensures** that appropriate string of flights with their actual or predicted takeoff times are now available in the prediction simulation queue "m.q". The operation also sets the ready_to_extract flag to true indicating that the simulation results are available. In the ensures clause, #m denotes the value of parameter m before the call and m denotes its value after the call.

The effect of the Simulate_Runway operation has been specified formally using a mathematical definition Proper_Flights_In_ Queue. This definition uses another definition Well_Formed_Runway_Queue that specifies when a mathematical string (or sequence) of flight/take-off time pair is a valid simulation, based on database information such as push back times, taxi times, and runway delays. The definition Proper_Flights_In_Queue is additionally concerned with a given window of time. Both of these definitions, as well as others not listed explicitly in the figure, are based on departure database information. The relevant database details are contained in db, modeled by SMA_DEPARTURE_DB_MODEL. Details of this model are the topic of the next subsection.

The Extract_Next operation requires that the results be ready for extraction, and it **produces** the predicted values of the next flight number and associated takeoff time. Cumulative_Wait_Time operation returns the total wait time of the flights in line (as specified formally in the mathematical operation Cumulative_Wait_Time_Def). The Queue_ Length_Of operation returns the number of flights left in the prediction simulation window of time.

A Mathematical Interface of COTS database

It is clear that the specification in Figure 2 can be meaningful only if there is a suitable mathematical modeling of database information. The mathematical description of SMA_DEPARTURE_DB_MODEL should contain all information relevant for predicting take-off times, but nothing more. Such a description is given in the interface of the **mathematics** module SMA_Database_Math_ Machinery in RESOLVE notation in Figure 3.

Unlike a concept specification, such as the one in Figure 2, that provides program types and operations to manipulate programming objects, the purpose of a mathematics module is to define mathematical types and definitions useful for writing specifications. Mathematics modules are not implemented. They simply establish formal meanings for domain vocabulary. (Mathematics modules in RESOLVE are similar in spirit to Larch traits [Wing 90].)

In Figure 3, SMA_DEPARTURE_DB_MODEL is defined to be a **math subtype**. A math subtype is essentially a base mathematical type with zero or more constraints on the value space [Heym 94, Rushby 98]. In the definition of SMA_DEPARTURE_DB_MODEL, other math subtypes have been defined and used, though most of them have not been shown. This database model consists of a collection of functions that are needed in the prediction system. For example, predicted_pushback_time is a function from the character string FLIGHT_ID into DISCRETE_TIME. In this case, the base mathematics type for DISCRETE_TIME is integer, where the values are constrained to be non-negative integers.

Some of the information in the database is particular to the specific airport. This information has been isolated and specified separately in another mathematics module SMA_Database_Airport_Information (not shown). This module contains information such as airport-specific gate names, flight identifications, and standard splits. Isolating the airport specific information in this one module enhances the portability of this system.

Together the mathematical modules define *mathematical interface contracts* of the COTS product. For the prediction system, any database can be used as long as it contains at least the information corresponding to the mathematical modeling in Figure 3. The example illustrates that in general, formal specification of COTS-based systems may require selected aspects of the COTS products to be modeled mathematically and captured formally.

```
mathematics SMA_Database_Math_Machinery
  context
    global context
      mathematics SMA_Database_Airport_Information
  interface
    math subtype FLIGHT_ID is string of character
      exemplar fid
      constraint Is_Allowed_Flight_ID(fid)

-- similar math subtypes for AIRCRAFT_TYPE_NAME,
-- GATE_NAME, OPTIONAL_GATE_NAME, DGA_NAME,
-- RUNWAY_ID, OPTIONAL_RUNWAY_ID,
-- and SPLIT_ID

    math subtype GATE_RUNWAY_PAIR is (
        g: GATE_NAME,
        r: RUNWAY_ID )

    math subtype CONFIGURATION is (
        sid: SPLIT_ID,
        split: function from DGA_NAME to RUNWAY_ID)

    math subtype FLIGHT_TIME_PAIR is (
        f: FLIGHT_ID,
        t: DISCRETE_TIME)

    math subtype SMA_DEPARTURE_DB_MODEL is (
      aircraft_type: function from FLIGHT_ID to
        AIRCRAFT_TYPE_NAME
      gate: function from FLIGHT_ID to
        OPTIONAL_GATE_NAME
      departure_gate_area: function from FLIGHT_ID to
        DGA_NAME
      runway_used: function from FLIGHT_ID to
        OPTIONAL_RUNWAY_ID
      predicted_pushback_time: function from FLIGHT_ID
        to DISCRETE_TIME
      actual_pushback_time: function from FLIGHT_ID to
        OPTIONAL_DISCRETE_TIME
      actual_takeoff_time: function from FLIGHT_ID to
        OPTIONAL_DISCRETE_TIME
      delay_time: function from AIRCRAFT_TYPE_NAME
        to DISCRETE_TIME
      roll_time: function from AIRCRAFT_TYPE_NAME
        to DISCRETE_TIME
      taxi_time: function from GATE_RUNWAY_PAIR to
        DISCRETE_TIME
      configuration: set of CONFIGURATION )

end SMA_Database_Math_Machinery
```

Figure 3: Mathematical Modeling of Database Information

4. COTS-Based Implementation and Modular Reasoning of the Subsystem

This section describes a component-based implementation of the prediction subsystem. To be able to reason in a modular fashion about a COTS-based implementation, it is essential to have a *programmatic interface contract(s)* for the COTS product. This is the topic of this section.

In *modular reasoning* it is possible to ensure that a component implementation satisfies its specification, based only on the specifications of reused components [Ernst 90, Leavens 91, Weide 94b]. Figure 4 provides an illustration to explain the basic idea. In the figure, an oval represents the specification of a component, whereas a rectangle represents an implementation. A thin arrow labeled "i" indicates an *implements* relationship and a thick arrow labeled "u" indicates a *uses* relationship.

To verify that the implementation in the figure satisfies its specification, only the specifications of reused components, numbered 1, 2, and 3 are needed. No knowledge of the rest of the system in which the component will be used is necessary and no knowledge of details of implementations of reusable components is necessary. Modular reasoning essentially makes it possible to reason about one implementation at a time, and is therefore, scalable. This basic idea is independent of whether the reasoning process is formal or informal, automated or manual.

Figure 5 contains a COTS-based implementation of the flight departure prediction subsystem. The implementation uses the specification of a part of the COTS database (as well as other supporting specifications). In the figure, the database interface has been separated into two concepts, one that specifies basic database retrieval operations and another that specifies a highly application-specific extensive database query. These interfaces need to be implemented using COTS software and they serve as the programmatic contract between the prediction subsystem and COTS database. (Mathematics modules are not shown in Figure 5.)

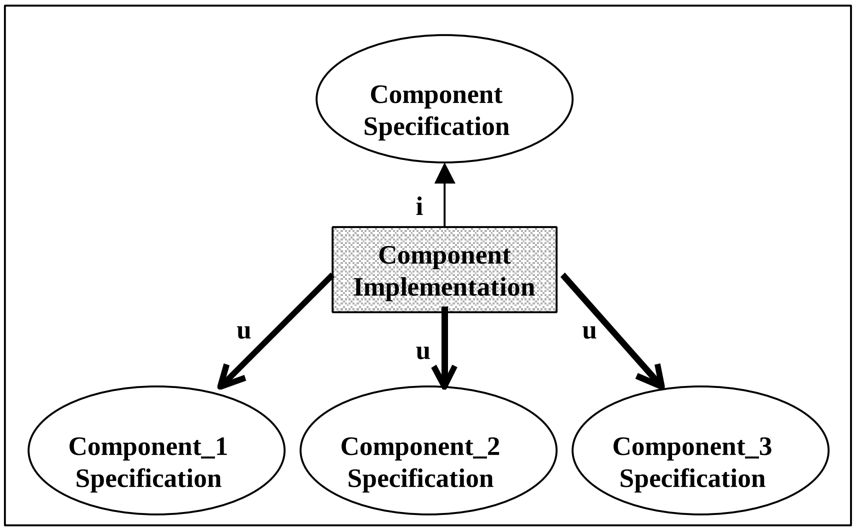

Component Specification
i
Component Implementation
u
u
u
Component_1 Specification
Component_2 Specification
Component_3 Specification

Figure 4: Illustration to Explain Modular Reasoning

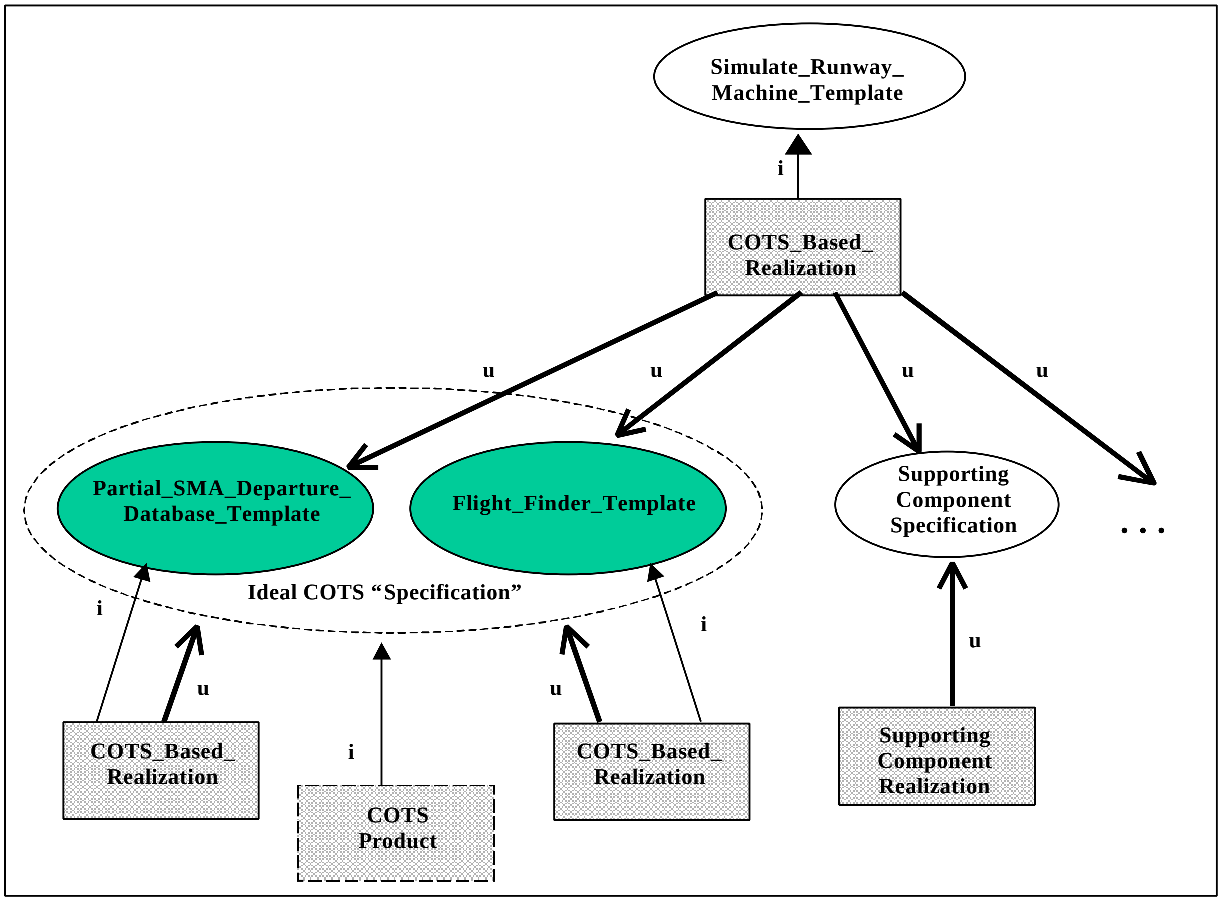

Simulate_Runway_ Machine_Template
i
COTS_Based_ Realization
u
u
u
u
Partial_SMA_Departure_ Database_Template
Flight_Finder_Template
Supporting Component Specification
. . .
Ideal COTS "Specification"
i
i
u
u
u
i
COTS_Based_ Realization
COTS Product
COTS_Based_ Realization
Supporting Component Realization

Figure 5: Modular Reasoning for the SMA Flight Departure Prediction Algorithm

Figures 6 and 7 contain programmatic database interface contracts (or specifications) for Partial_SMA_Departure_Database_Template and SMA_Database_Flight_Finder_Template.

Partial_SMA_Departure_Database_Template is shown in Figure 6. This programmatic interface depends on the mathematical interface shown in Figure 3 in the previous section. The global context explains this linkage by referring to SMA_Database_Math_Machinery In the interface section, the Database object is modeled by SMA_DEPARTURE_ DB_MODEL (from Figure 3). The operations specify basic retrieval operations of the database.

The other part of the Database interface, SMA_Database_Flight_Finder_Template, is shown in Figure 7. This interface specifies an extended database query, Select_Flights_To_ Runway, for simulating the runway. The query locates all flights that either have used the specified runway or will use the runway based on the specified split, and orders them by takeoff time (if they have already departed) or time of pushback from the gate (actual pushback times followed by predicted pushback times). The exported type is the conceptual value of the result of this selection query. The other two operations are used to return the next flight ID from the ordered selection, and to check on the number of flights remaining from the selection.

Together, the interfaces in Figures 6 and 7 accurately capture programmatic aspects of the COTS product, relevant to the SMA departure prediction subsystem.

```
concept Partial_SMA_Departure_ Database_Template

  context global context

    mathematics SMA_Database_Math_Machinery

  interface
    type SMA_Database is modeled by
      SMA_DEPARTURE_DB_MODEL
      exemplar db

    operation Aircraft_Type (
        preserves db: SMA_Database_Machine,
        preserves fid: Char_String): Char_String
      requires Is_Allowed_Flight_ID (fid)
      ensures db.aircraft_type(fid) = Aircraft_Type

    operation Taxi_Time (
        preserves db: SMA_Database_Machine,
        preserves gid: Char_String,
        preserves rid: Char_String): Integer
      requires Is_Allowed_Flight_ID (fid)
        and Is_Allowed_Gate_Name (gid)
      ensures db.taxi_time ((gid, rid)) = Taxi_Time

-- similar operations to obtain the Gate, Runway_Used,
-- Departure_Gate_Area, Predicted_Pushback_Time,
-- Actual_Pushback_Time, and Actual_Takeoff_Time
-- associated with a flight ID; the Delay_Time and
-- Roll_Time associated with an aircraft type; and the
-- Assigned_Runway associated with a split and DGA

end Partial_SMA_Departure_Database_Template
```

Figure 6: Database Interface Contract (part 1)

```
concept SMA_Database_Flight_Finder_Template

  context

    global context

      facility Standard_ Partial_SMA_Departure_
        Database_Facility

    local context
      math operation
        All_Flights_To_Runway_In_Pushback_Order (
          q: string of FLIGHT_ID,
          sid: SPLIT_NAME,
          rid: RUNWAY_NAME,
          db: SMA_DATABASE_MACHINE): boolean
      explicit definition
        all the flights that have taken off from the specified
        runway or that are assigned to the specified runway
        by the split are  in the string, and the flights are in
        order of (by precedent)
          (1) actual takeoff time (if the flight has taken off),
          (2) actual pushback time
              (if the flight has pushed back from the gate but
               has not taken off),
          (3) predicted pushback time
              (if the flight has not pushed back from the gate).

  interface
    type Flight_Finder_State is modeled by (
        q: string of FLIGHT_ID)
      exemplar ff
      initialization ensures    |ff.q| = 0

    operation Select_Flights_To_Runway (
        alters   ff: Flight_Finder_State,
        preserves sid: Char_String,
        preserves rid: Char_String,
        preserves db: SMA_Database_Machine)
      requires  Is_Allowed_Split_Name (sid) and
        Is_Allowed_Runway_Name (rid)
      ensures
        All_Flights_To_Runway_In_Pushback_Order
        (ff.q, sid, rid, db)

    operation Get_Next_Flight_To_Runway (
        alters   ff: Flight_Finder_State,
        produces  fid: Char_String)
      requires   |ff.q| > 0
      ensures #ff.q = <fid> * ff.q

    operation Number_Of_Flights_To_Runway (
        preserves ff: Flight_Finder_State,) : Integer
      ensures    Number_Of_Flights_To_Runway = |ff.q|

end SMA_DB_Departure_Get_Flights_Template
```

Figure 7: Database Interface Contract (part 2)

5. Discussion

Status

The specification for the SMA subsystem has been implemented in RESOLVE/C++. [Hollingsworth 94] The implementations of the database concepts depend on the particular database product and its structure, and consist primarily of Structured Query Language (SQL) queries.

The shadow development effort, on which this case study is based, uses an mSQL database rather than the Oracle database that is used by the developer. The implementations for an mSQL database have substantial differences from implementations using Oracle, but each implementation should meet the mathematical and programmatic interface contracts. A change in the choice of the underlying COTS database affects only the implementations of the database interface concepts. No other specifications or implementations will be impacted by changing the COTS package, or by changing the physical storage within the database instance.

A separate part of the application instantiates Simulate_Runway_Machine_Template, sets the runway, split, and times, and then obtains cumulative wait time for the runway and the predicted takeoff times for the flights. This part of the software is responsible for storing the data as necessary in the database and for displaying the data (hence there are no operations in these concepts to store or display data). The concepts and implementations are suitable for any airport. The only changes are relegated to an airport-specific data module.

Related Work

While there is much work on developing COTS-based software and formal methods in the reuse community, few concrete efforts to bridge the gaps exist. A significant part of COTS work is on evaluation, although there is no accepted standard of evaluation [Carney 97, FAA 96] and much of the evaluation consists of comparing likely COTS candidates [Oberndorf 97]. COTS evaluation extends well beyond the functional evaluation of the current version of the COTS product, and includes such issues as training available on the product, stability of the vendor, and sustaining engineering support. [Parra 97] Formal methods transition techniques in supporting component certification [Leavens 98], component selection [Chen 97], and component modification [Jeng 94], but none of these efforts address COTS software.

The National Product Line Asset Center (NPLACE) has established a method of evaluating COTS products against a set of predefined testable criteria. Voas has developed a method for COTS certification that involves testing the product based on the operational profile of the system, system-level fault injection, and operational system testing. [Voas 98b] Voas also advocates taking defensive measures by putting a wrapper around the COTS software to limit the output of the COTS software to acceptable values. However, [Weyuker 98] discusses many difficulties in effectively testing components that are being used to construct a system, either as individual components or within the developed system. [Leach 97] suggests that the best approach may be to locate a COTS vendor that can be trusted, and attempt to match the interface specifications.

The specifications of database interface concepts in this paper serve as a guide to testing the COTS product. They can be used to generate test cases to provide assurance that the COTS product satisfies the specifications. This approach limits testing on the COTS product to those features of the COTS product that are actually used in the application being developed, rather than attempting to test all features of the COTS product.

Conclusions

This paper illustrates using a case study how formal methods can be applied to COTS-based software. Using a realistic case study, it illustrates the issues in specification and implementation of a safety-critical subsystem. The approach allows developing mathematical and programmatic interface contracts for only selected aspects of the system.

The interface contracts isolate the system under development from the COTS software. Changes and improvements to COTS software products have no impact on the system; where there is an impact it is seen through the interfaces. The interface contracts facilitate formal reasoning and also serve as a guide to the test process to validate the use of the COTS software.

Acknowledgements

We are pleased to thank the members of the Reusable Software Research Group at the West Virginia University for numerous discussions relating to the topics of this paper. We are also pleased to acknowledge our financial sponsors: DARPA grant DAAH04-96-1-0419, monitored by the U. S. Army Research Office and NASA Cooperative Agreement NCC 2-979 at the NASA/Ames IV&V Facility in Fairmont, WV.

References

[Carney 97] David Carney and Patricia Oberndorf, "The Commandments of COTS: Still in Search of the Promised Land," *Crosstalk*, Vol 10, No 3, May 1997.

[Chen 97] Yonghao Chen and Betty H. C. Cheng, "Formalizing and Automating Component Reuse," *Proceedings of the 9th International Conference on Tools with AI*, IEEE, November 1997.

[Ernst 90] Ernst, G. W., Hookway, R. J., Menegay, J. A., and Ogden, W. F., "Modular Verification of Ada Generics", *Computer languages 16*, 3/4, 1991, pp. 259-280.

[FAA 96] "Use of COTS/NDI in Safety-Critical Systems," *Challenge 200 Subcommittee of the FAA Research*, Engineering, and Development Advisory Committee, February 1996.

[Heym 94] Wayne Heym, et. al., *Mathematical Foundations and Notation of RESOLVE*, Technical Report OSU-CISRC-8/94/TR45, revised September 1998, Department of Computer and Information Science, The Ohio State University, Columbus, OH.

[Hollingsworth 94] Joseph E. Hollingsworth, et. al., "RESOLVE Components in Ada and C++," *Software Engineering Notes*, Vol 19, No 4, October 1994.

[Jeng 94] Jun-Jeng and Betty H. C. Cheng, "A Formal Approach to Reusing More General Components," *Proceedings of the 9th Knowledge-Based Software Engineering Conference*, IEEE, September 1994

[Leach 97] Ronald J. Leach, *Software Reuse*, McGraw-Hill, New York, 1997.

[Leavens 91] Leavens, G., "Modular Specification and Verification of Object-Oriented Programs", *IEEE Software*, Vol. 8, No. 4, July 1991, pp. 72-80.

[Leavens 98] Gary T. Leavens, Oscar Nierstrasz, and Murali Sitaraman, "1997 Workshop on Foundations of Component-Based Systems," *Software Engineering Notes*, Vol 23, No 1, January 1998.

[Oberndorf 97] Patricia A. Oberndorf, et. al., *Workshop on COTS-Based Systems*, Software Engineering Institute, CMU/SEI-97-SR-109, November 1997.

[Parra 97] Amalia Parra, et. al., "Packaged-Based Development Process in the FDD," *Proceedings of the Software Engineering Workshop*, NASA/Goddard Space Flight Center, December 1997.

[Rushby 98] John Rushby, Sam Owre, and N. Shankar, "Subtypes for Specifications: Predicate Subtyping in PVS," *IEEE Transactions on Software Engineering*, Vol 24, No 9, September 1998.

[SMA 95] *Surface Movement Advisor Build 1 System Requirements Document*, SMA-110, National Aeronautics and Space Administration, Ames Research Center, June 1995.

[Sitaraman 94] Murali Sitaraman and Bruce Weide, "Component-Based Software using RESOLVE," *Software Engineering Notes*, Vol 19, No 4, October 1994.

[Voas 98a] Jeffrey M. Voas, "The Challenges of Using COTS Software in Component-Based Development," *IEEE Computer*, Vol 31, No 6, June 1998.

[Voas 98b] Jeffrey M. Voas, "Certifying Off-the-Shelf Components," *IEEE Computer*, Vol 31, No 6, June 1998.

[Weide 94a] Bruce W. Weide, William F. Ogden, and Murali Sitaraman, "Recasting Algorithms to Encourage Reuse," *IEEE Software*, Vol 11, No 5, September 1994.

[Weide 94b] Weide and Hollingsworth, *On Local Certifiability of Software Components*, Technical Report OSU-CISRC-1/94-TR04, Dept. of Computer and Information Science, The Ohio State University, 1994.

[Weyuker 98] Elaine J. Weyuker, "Testing Component-Based Software: A Cautionary Tale," *IEEE Software*, Vol 15, No 5, September/October 1998.

[Wing 90] Wing, J. M., "A Specifier's Introduction to Formal Methods", *IEEE Computer*, 29(9), September 1990, 8-24.